GCC后端中四路双精度短向量寄存器的实现

李春江 杜云飞 倪晓强 王永文 杨灿群
(国防科学技术大学计算机学院 长沙 410073)

摘 要 设计和实现一个新的产品化的编译器通常需要几年时间。基于已有的编译器进行修改和扩展,是研发面向新体系结构的编译器的主要途径。GNU编译器集合(GCC)支持多种高级语言和多种目标处理器平台、文档及源代码开放等。基于GCC的Sparc后端,实现了支持四路双精度SIMD指令的四路双精度短向量寄存器的描述。在此过程中,定义了新的目标机,扩充了一类向量模式,定义了一类新的寄存器约束,实现了四路双精度寄存器的描述,定义了四路双精度SIMD指令的机器描述。对于面向此类SIMD指令的内嵌函数,GCC编译器能够正确使用该类向量寄存器来生成对应的SIMD指令。

关键词 GCC后端,四路双精度,向量寄存器
中图法分类号 TP314 **文献标识码** A

Implementation of Four-way Double Precision Short Vector Registers in GCC Backend

LI Chun-jiang DU Yun-fei NI Xiao-qiang WANG Yong-wen YANG Can-qun
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract It will cost several years to design and implement a new product-level compiler. Designing and implementing based on an already-issued product-level compiler are the main approach to develop a compiler for a new architecture. GNU compiler collection (GCC) supports multiple high level languages and multiple platforms, and its internal documents and source code are open. Based on the Sparc backend of GCC, we implemented the description of four-way double-precision short vector registers which support four-way double-precision SIMD instructions. In this process, we defined a new target machine, expanded a new vector mode, defined a new class of register constraints, provided the descriptions of four-way double-precision short vector registers, designed the machine descriptions of the four-way double-precision SIMD instructions. For the builtin functions for this kind of SIMD instructions, our GCC can produce correct SIMD instructions using such kind of vector registers.

Keywords GCC backend, Four-way double precision, Vector registers

1 背景

1.1 GCC编译器集合

GCC(Gnu Compiler Collection,Gnu编译器集合(下文简称GCC或GCC编译器))[1]是广泛使用的开源编译器套件。近10余年来,随着Linux操作系统、GNU二进制工具的广泛使用,GCC编译器已经成为Linux平台上必备的编译器,发展非常迅速。对GCC发展的贡献主要来自两个方面:其一是开源社区;其二是产业界重要的公司,如Redhat,IBM,Intel等。所有基于GCC的开发工作都必须遵循GPL(Gnu Public Licensee)[2]规则,其中最重要的是必须保证对GCC所做的改动也开放源代码。

GNU编译器具有支持多种语言和多种目标处理器平台、文档及源代码开放等特点。基于GCC编译器面向新的语言,新的平台进行的编译器开发和编译优化工作非常活跃。

1.2 自主通用处理器对GCC的需求

面向处理器的编译系统和工具链是和处理器关联最为密切的系统软件。若干年以来,国家科研计划项目一直持续支持自主通用处理器的研究开发。国内自主处理器的研制单位无一例外地都非常重视GCC编译器对各自研制的处理器的支持。有些研制单位提供的编译系统和工具链完全基于GCC和GNU工具链实现;有些单位即使其主要的编译器并非基于GCC开发,但仍要在GCC中实现对其自主处理器的支持(要么自己独立基于GCC实现,要么求助于第三方软件开发单位或个人基于GCC实现)。总之,基于GCC编译系统实现一套面向自主通用处理器的编译系统已经成为所有研制

到稿日期:2011-11-01 返修日期:2012-03-02 本文受国家自然科学基金项目(61170046,61170045)资助。

李春江(1974-),男,博士,副研究员,主要研究方向为计算机体系结构、编译及优化技术,E-mail:chunjiangli@gmail.com;杜云飞(1980-),男,博士,助理研究员,主要研究方向为编译技术、系统软件;倪晓强(1975-),男,博士,副研究员,主要研究方向为处理器体系结构及微电子技术;王永文(1977-),男,博士,副研究员,硕士生导师,主要研究方向为处理器体系结构;杨灿群(1968-),男,博士,研究员,硕士生导师,主要研究方向为编译技术、系统软件。

单位必须完成的工作。

2 目标平台体系结构

基于 OpenSparc T2 处理器内核[3]，设计了 FTXX 处理器，一个主要的增强是设计了支持四路双精度 SIMD(Single Instruction Multiple Data，单指令多数据)处理的 VPU(Vector Processing Unit，向量处理单元)，并在原有指令集的基础上，设计和实现了相应的 SIMD 指令。设计和实现向量长度更长、支持双精度数值计算的 SIMD 部件，是近年来通用高性能微处理器的发展趋势之一。如 Intel 的 AVX[4]已经支持四路双精度向量的 SIMD 并行处理。

因此，FTXX 处理器对编译系统和工具链提出了新的实现和优化要求。按 GCC 中支持 SIMD 指令的一般处理方法，首先是要实现面向 SIMD 指令的内嵌函数(Builtin Functions)，为在 C 语言程序中使用 SIMD 指令提供一类函数级 API；其次是在编译器中实现面向 SIMD 指令的自动向量化和性能优化(这涉及到别名分析、相关性分析、自动向量化算法、指令调度、访存优化等很多方面的问题)。这些编译器实现和优化工作中最基本的一条是要在编译器后端中能够正确描述这些 SIMD 指令。

FTXX 处理器的 VPU 支持四路双精度数据的 SIMD 处理，其寄存器长度为 256 位，包含 4 个双精度浮点数据。在 GCC 后端中扩展对此类向量寄存器的支持是 GCC 后端实现中的重要内容。

2.1 处理器内核结构

图 1 是 FTXX 处理器内核结构框图。

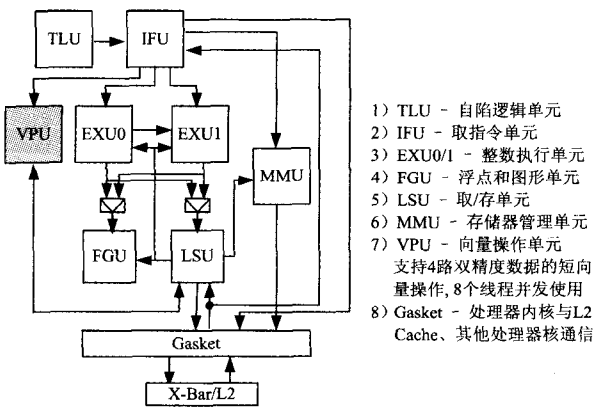

图 1 FT1500 处理器内核体系结构

该处理器内核中的 VPU 支持四路双精度数据的 SIMD 处理。

OpenSparc T2 处理器内核采用轮转多线程的方式支持 8 个硬件线程，有两条整数流水线，每 4 个线程共享一条整数流水线；有一条浮点流水线，8 个线程共享。因此，原有的处理器内核的浮点计算能力相对较弱。在处理器内核中扩展支持四路双精度计算的 VPU，就是为了提升处理器对数据密集的双精度数据的数值计算能力。

2.2 短向量寄存器

向量寄存器位于 VPU 中，用来存储 8 个线程的四路双精度 SIMD 数据。向量寄存器的大小为 8×32×256 位，按照每个线程 1 组，每组 32 个 256 位的寄存器构成，如图 2 所示。

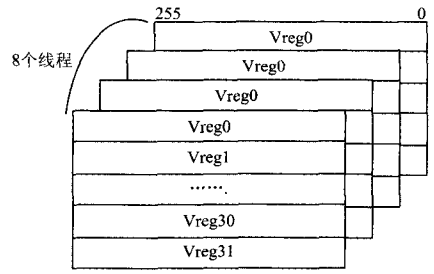

图 2 向量寄存器逻辑视图

3 指令集概况

这里仅列出使用四路双精度短向量寄存器的部分指令，如表 1 所列。

表 1 使用四路双精度短向量寄存器的一些指令

数据移动类	STV	stv $vfreg_{rs}$,[address]	把 4 个 64 位的双精度数据从向量寄存器 VF[rc]写到存储器中
	LDV	ldv [address], $vfreg_{rd}$	把 4 个 64 位的双精度数据从存储器装载到向量寄存器 VF[rd]中
	LD1TO4	ld1to4 [address], $vfreg_{rd}$	将 1 个 64 位的双精度数据从存储器装载到向量寄存器 VF[rd]的 4 个位置中
	FVMOV	fvmov $vfreg_{rs}$, $vfreg_{rd}$	把 VF[rs]中的 4 个 64 位双精度数据赋值给 VF[rd]
数值运算类	FVADD	fvaddd $vfreg_{rs1}$, $vfreg_{rs2}$, $vfreg_{rd}$	四路双精度数据的四则运算
	FVSUB	fvsub $vfreg_{rs1}$, $vfreg_{rs2}$, $vfreg_{rd}$	
	FVMUL	fvmul $vfreg_{rs1}$, $vfreg_{rs2}$, $vfreg_{rd}$	
	FVDIV	fvdiv $vfreg_{rs1}$, $vfreg_{rs2}$, $vfreg_{rd}$	
数值转换类	FVdTOs	fvdtos $vfreg_{rs}$, $vfreg_{rd}$	把向量寄存器 VF[rs]中 4 个双精度数据转换成单精度浮点后，存到 VF[rd]中
	FVsTOd	fvstod $vfreg_{rs}$, $vfreg_{rd}$	把向量寄存器 VF[rs]中 4 个单精度数据转换成双精度浮点后，存到 VF[rd]中
数值设置类	SETVF0	setvf0 $vfreg_{rd}$	将浮点+0 存入向量寄存器 VF[rd]中
	SETVF1	setvf1 $vfreg_{rd}$	将浮点+1 存入向量寄存器 VF[rd]中
数值比较类	FVCMP	fvcmp $vfreg_{rs1}$, $vfreg_{rs2}$	把向量寄存器 VF[rs1]和 VF[rs2]中的值进行比较，结果影响状态寄存器

4 向量寄存器扩展

4.1 目标机、寄存器约束和机器模式的定义

4.1.1 目标机定义

在 Sparc 后端的 sparc.opt 文件中加入如下 3 行：

```
mvpu
Target Report Mask(VPU)
Use hardware VPU in FTXX
```

如此定义以后，GCC 编译器就可以接受“-mvpu”选项，并当此选项有效时，目标机标识“TARGET_VPU”被定义，在编译器中根据此目标机定义进行编译控制。

4.1.2 寄存器约束定义

在 Sparc 后端的 constraints.md 中加入如下内容：

```
(define_register_constraint "v"
```

"(TARGET_VPU ? FTXX_VEC_REGS: NO_REGS)"
"Any vector register in VPU mode")

如此定义后，在机器描述中用“v”字符就可以限定寄存器为面向“TARGET_VPU”的四路双精度寄存器，该类寄存器定义为“FTXX_VEC_REGS”。

4.1.3 机器模式定义

在 Sparc 后端的 sparc-modes.def 文件中加入如下 1 行：

VECTOR_MODES (FLOAT,32);

如此定义后，Sparc 后端中就新增加了 1 类向量模式，该模式的含义是：32 个字节的向量，包含 4 个双精度浮点数。

4.2 四路双精度短向量寄存器的实现

4.2.1 在 Sparc 后端的 sparc.h 文件中，面向向量寄存器扩展所做的修改

1)重新定义每个 SIMD 字所包含的单元数

#define UNITS_PER_SIMD_WORD(MODE) (TARGET_VPU && ((MODE) == V4DFmode) ? 32:(TARGET_VIS ? 8:UNITS_PER_WORD))

含义是：在定义了“TARGET_VPU”且机器模式为“V4DFmode”时，每个 SIMD 字包含的单元(即字节)数为 32，否则保持原来 Sparc 后端中的定义。原来的 Sparc 后端为支持 VIS[5] 指令，定义 UNITS_PER_SIMD_WORD 为 8，它将浮点寄存器用作 SIMD 寄存器，仅支持整型数据的 SIMD 处理。

2)重新定义最大的存储器对齐约束

使存储器对齐约束变为支持 4 路双精度 SIMD 短向量的 256 位对齐。

3)重新定义第一个伪寄存器编号

因为面向 VPU 增加了 32 个短向量寄存器和 1 个 VPU 状态寄存器，所以第一个可以用作伪寄存器编号的值就增大了 33。

4)增加寄存器编号宏定义及寄存器类型判定宏定义

用此宏定义可以根据硬件寄存器编号判定是否属于四路双精度短向量寄存器。

5)修改 FIXED_REGISTERS 数组定义

在 FIXED_REGISTERS 定义中增加 32 个向量寄存器和 1 个状态寄存器的描述：

```
#define FIXED_REGISTERS \
{1,0,2,2,2,2,1,1,          \
 0,0,0,0,0,0,1,0,          \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,1,1,          \
                           \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,0,0,          \
                           \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,0,0,          \
 0,0,0,0,0,0,0,0,          \
                           \
 0,0,0,0,0,1,1,1,          \
 1,1,1,1,1,1,1,1,          \
 1,1,1,1,1,1,1,1,          \
 1,1,1,1,1,1,1,1,          \
                           \
 1,1,1,1,1,1,1,1           \
}
```

该数组是目标平台钩子(Target Hook)，将被赋值给 fixed_regs 数组；后面的 33 个 1 代表面向 VPU 新增加的 33 个硬件寄存器(1 个 VPU 状态寄存器和 32 个四路双精度短向量寄存器)。该数组的内容根据不同目标平台可改。1 表示不用于寄存器分配。

上述定义的第一行表示全局寄存器 g0－g7，g0 为常整数 0，不用于寄存器分配；全局寄存器 g2－g4 为应用保留；全局寄存器 g6－g7 为操作系统保留，在嵌入式环境下为应用保留；g1 和 g5 用于临时寄存器使用。这些定义随目标机的不同在编译器实现中进行更改。

6)修改 CONDITIONAL_REGISTER_USAGE 宏定义

该宏定义也是一个目标平台钩子，它根据不同目标机的情况(64 位、32 位、嵌入式处理器等)，对 fixed_regs 数组进行修改。当目标平台为 TARGET_VPU 时，将对应硬寄存器编号的 fixed_regs 数组内容改为 0，表示可用于寄存器分配。FTXX_VPUS_REG 已经定义为 102 号，即第 102 号硬件寄存器是 VPU 状态寄存器。

7)修改 CALL_USED_REGISTERS 宏定义

该宏定义指向一个数组，数组中的每一个元素表示一个硬件寄存器。数组中为 1 的元素表示不用于函数调用，即函数调用中不用作参数传递。该数组后半部分的 33 个“1”是面向 VPU 新增加的。

8)修改 HARD_REGNO_NREGS(REGNO,MODE)宏

该宏的含义是对于硬件寄存器编号 REGNO 和机器模式 MODE，定义需要几个寄存器作为一个整体，即一个寄存器使用。由于历史演进的原因，Sparc 平台支持双精度浮点数的寄存器需要连续两个原来存储单精度浮点数据的寄存器作为一个双精度数据寄存器用，即双精度浮点数据仅能用 f0，f2 等偶数编号的寄存器。针对 FTXX 处理器的 VPU 单元，当 TARGET_VPU 是四路双精度短向量寄存器时，那么就用一个该类型的短向量寄存器存储该类型的数据。

9)修改 REGMODE_NATURAL_SIZE(MODE)宏

该宏定义的含义是机器模式所对应的字节数。修改后的效果是当机器模式为四路双精度短向量寄存器所支持的 V4DF 时，寄存器的长度为 32 个字节。

10)定义寄存器类枚举类型及相应的寄存器类所对应的名字

在 reg_class 枚举类型和 REG_CLASS_NAMES 字符串数组中分别增加了 VPUS_REG 和 FTXX_VEC_REGS 及对应的名字，其表示 VPU 的状态寄存器和四路双精度短向量寄存器。

11)定义每类寄存器对应哪些硬件寄存器

将 REG_CLASS_CONTENTS 定义修改为：

```
#define REG_CLASS_CONTENTS \
{ {0,0,0,0,0},\
 {0,0,0,0xf,0},\
```

```
{0,0,0,0x40,0},\
{0xffff,0,0,0,0},\
{-1,0,0,0x20,0},\
{0,-1,0,0,0},\
{0,-1,-1,0,0},\
{-1,-1,0,0x20,0},\
{-1,-1,-1,0x20,0},\
{0,0,0,0xffffff80,0x7f},\
{-1,-1,-1,-1,0x7f} }
```

这个定义的组织很有特点，它用 5 个整数(32 位整数)数组中的各个位对应一个硬件寄存器，可以和 GCC 寄存器分配中用的位集合数组变量 hard-reg-set 对应。在上述定义中，面向 VPU 增加了 VPUS_REG 和 FTXX_VEC_REGS 所对应的位描述。

12)定义需要用集成寄存器分配器(Integrated Register Allocator)[6]分配的寄存器类

定义如下：

```
#define IRA_COVER_CLASSES \
{ GENERAL_REGS, EXTRA_FP_REGS, FPCC_REGS, FT1500_VEC_REGS,\
VPUS_REG, LIM_REG_CLASSES }
```

该定义也是一个目标平台钩子，GCC 编译器的寄存器分配阶段据此来判断哪些类寄存器需要用 IRA 来分配。

13)修改寄存器分配的顺序

寄存器分配顺序定义在字符串数组 REG_ALLOC_ORDER 中，面向 FTXX 处理器的 VPU 单元增加了向量寄存器名字和 VPU 状态寄存器名字，在编译器生成的汇编程序中将使用这些寄存器名字。

4.2.2 *在 Sparc 后端的 sparc.c 文件中，面向四路双精度短向量寄存器所做的修改*

1)修改 leaf_reg_remap 数组定义

该数组定义了函数调用中输入寄存器映射到输出寄存器的关系。仅函数调用中用于参数传递的寄存器和用作硬栈桢指针的寄存器映射关系会改变，其它寄存器保持原来的对应关系。该数组中元素 102－134 号寄存器是面向 VPU 新增加的，由于这些寄存器不用于参数传递和保存桢指针，因此映射关系不变。

2)修改 sparc_leaf_regs 数组定义

该数组和 leaf_reg_remap 数组一样，都是用硬件寄存器号来索引的。该数组中为 1 的元素所对应的硬件寄存器可以由被调用函数操作。

3)扩展模式类枚举类型定义

在 sparc_mode_class 枚举类型中增加了 VPUS_MODE 和 V_MODE，它们分别表示面向 VPU 的状态寄存器和向量寄存器。

4)定义向量模式标识

```
#define V_MODES (1<<(int) V_MODE)
#define VF_MODES (V_MODES)
```

根据枚举值 V_MODE 定义向量模式标识 V_MODES。定义 VF_MODES，表示双精度浮点向量。

5)修改每个硬件寄存器对应的模式值的内容

在数组 hard_64bit_mode_classes 中增加后面 33 个硬件寄存器对应的机器模式值，状态寄存器为 VPUS_MODES，短向量寄存器为 VF_MODES。该数组以硬件寄存器号索引，定义了每个硬件寄存器的模式值。

6)在模式初始化函数中初始化各个硬件寄存器对应的模式

在 static void sparc_init_modes (void)函数中，增加如下内容：

```
case MODE_VECTOR_FLOAT:
  if (i == V4DFmode)
sparc_mode_class[i]=1<<(int) V_MODE;
  break;
```

其含义是定义机器模式对应的标识值，并且在该函数根据硬件寄存器编号确定寄存器类型的循环中，加入以下内容：

```
else if ((i > 102) && (i < 135))
sparc_regno_reg_class[i]=FTXX_VEC_REGS;
```

其含义是为向量寄存器定义寄存器类。

7)修改向量模式判定函数

将函数 sparc_vector_mode_supported_p 的函数体改为：

```
return (TARGET_VIS || TARGET_VPU) && VECTOR_MODE_P (mode) ? true:false;
```

其含义是对于 VIS 和 VPU 目标机，当机器模式为向量模式时返回真。

5 达到的效果

通过上述面向四路双精度短向量寄存器的扩展工作，GCC 编译器能够正确地识别出处理相应的机器模式，并能正确完成寄存器分配和汇编指令生成。

如图 3 所示的用面向 4 路双精度短向量 SIMD 指令的内嵌函数编写的向量加程序，在 GCC 编译过程中，从树一级中间转换到 RTL[7]中间表示时，对应的向量加的指令如图 4(a)所示。

```
#include <stdio.h>
typedef double V 256 __attribute __((vector _size(32)));
#define N 1024
double src 1[N] __attribute __ ((aligned (32)));
double src 2[N] __attribute __ ((aligned (32)));
double res [N] __attribute __ ((aligned (32)));
int main ()
{
 int i;
 V256 res 1, res 2, res 3, res 4;
/* Initialization of source vector . */
 for (i = 0; i < N; i++)
 {
  src 1[i] = (double)i * 0.1;
  src 2[i] = (double)i * 0.1;
 }
/* Vector add using builtin functions . */
 for (i = 0; i < N/4; i++)
 {
  res 1 = __builtin _vpu _fvload (&src 1[i * 4]);
  res 2 = __builtin _vpu _fvload (&src 2[i * 4]);
  res 3 = __builtin _vpu _fvadd (res 1, res 2);
  __builtin _vpu _fvstore (&res[i * 4], res 3);
 }
 return 0;
}
```

图 3 内嵌函数编程举例

(下转第 306 页)

约了相关量子资源；通过将多目标扩展通用 Toffoli 门的条件动态设置为 0 和门电路的顺序级联设计方式，提高了比较器的运行效率，降低了出错率，增强了比较器的鲁棒性。

进一步的工作为：①进一步研究并优化量子比较器电路。②深化量子比较器的应用，将量子比较器应用到更多更复杂的算法中，利用量子计算的并行运算模式提高算法的性能。③设计更多的包括算术运算及逻辑运算在内的基本的专用量子电路，探讨量子计算机系统的构建模式。

参 考 文 献

[1] Nielsen M A, Chuang I L. Quantum Computation and Quantum Information [M]. Cambridge, Cambridge University Press, 2000

[2] Landauer R. Irreversibility and heat generation of the computing process [J]. IBM Journal of Research and Development, 1961, 5 (3): 183-191

[3] Deutsch D. Quantum theory, the Church-Turing principle and the universal quantum computer [J]. Proceedings of the Royal Society, 1985, 400(1818): 97-101

[4] Vedral V, Barenco A, Ekert A. Quantum networks for elementary arithmetic operations [J]. Physical Review A, 1996, 54(1): 147-153

[5] Bomble L, Lauvergnat D, Remacle A, et al. Controlled full adder or subtractor by vibrational quantum computing [J]. Physical Review A, 2009, 80(2): 022332/ 1-8

[6] Grover L K. Quantum mechanics helps in searching for a needle in a haystack [J]. Physical Review Letters, 1997, 79(2): 325-328

[7] Cheng S T, Wang C Y. Quantum switching and quantum merge sorting [J]. IEEE Transactions on Circuits and Systems, 2006, 53(2): 316-325

[8] Oliveira D S, Sousa P B, Ramos R V. Quantum search algorithm using quantum bit string comparator [C]// Proceedings of 2006 International Telecommunications Symposium. 2006: 582-585

[9] Oliveira D S, Ramos R V. Quantum bit string comparator: circuits and applications [J]. Quantum Computers and Computing, 2007, 7(1): 17-26

[10] Nascimento A L, Kowada L A B, Oliveira W R. A reversible ULA [C]// WECIQ: First Workshop-school in Quantum Information and Computation. Brazil, 2006

[11] Shigeru Y, Masaki N. An efficient framework to utilize Grover search [J]. Journal of Nanjing University of Posts and Telecommunications (Natural Science), 2011, 31(2): 49-58

[12] Khan M H A. Synthesis of quaternary reversible/quantum comparators [J]. Journal of Systems Architecture, 2008, 54(10): 977-982

[13] Arul A J. Impossibility of comparing and sorting quantum states [OL]. http://arxiv.org/abs/quant-ph/0107085, 2001

[14] Barcnco A, Bennett C, et al. Elementary gates for quantum computation [J]. Physical Review A, 1995, 52(5): 3457-3467

[15] Song Xiao-yu, Yang Guo-wu, Perkowski M, et al. Algebraic characteristics of reversible gates [J]. Theory of Computing Systems, 2005, 39(2): 311-391

[16] 李志强，陈汉武，徐宝文，等. 基于 Hash 表的量子可逆逻辑电路综合的快速算[J]. 计算机研究与发展，2008，45(12)：2162-2171

(上接第 295 页)

可见，在 RTX 中间语言级能够正确处理机器模式，并能正确分配伪寄存器。在 GCC 完成寄存器分配后，该指令的 RTL 描述如图 4(b)所示。可见，GCC 能够正确进行寄存器分配。最终生成的对应汇编指令为：fvaddd %v1，%v0，%v0。

```
(insn 89 88 90 6 example .c:21 (set (reg/v:V4DF 199 [ res3 ]) (plus:V4DF
(reg/v:V4DF 195 [ res1 ]) (reg/v:V4DF 198 [ res2 ]))) -1 (nil))
```

(a)

```
(insn 89 87 97 example .c:21 (set (reg/v:V4DF 103 %v0 [orig:199 res3 ]
[199])
    (plus:V4DF (reg:V4DF 104 %v1 [212])
        (reg:V4DF 103 %v0 [218]))) 504 {vpu_fvadd} (expr_list:REG_DEAD
(reg:V4DF 104 %v1 [212]) (nil)))
```

(b)

图 4 向量操作指令的 RTL 形式

结束语 GCC 编译器是一套非常庞大而复杂的软件系统。面向新目标机的指令集体系结构，在 GCC 后端中实现对该目标机的支持，是非常复杂的编译器工程实现工作。微软亚洲研究院的张亚勤博士说过“工程的能力决定创新的水平”。对具体的编译器所做的实现工作，在编译器工程技术领域非常重要。

基于 GCC 的 Sparc 后端，本文实现了支持四路双精度 SIMD 指令的四路双精度短向量寄存器描述。在此过程中，完成了新的目标机定义，扩充了一类向量模式，定义了一类新的寄存器约束，实现了四路双精度寄存器的描述。本文的工作对基于 GCC 进行的研究和开发工作有很大的参考价值。

参 考 文 献

[1] GCC, GNU Compiler Collection[OL]. http://gcc.gnu.org/

[2] The GNU General Public License[OL]. http://www.gnu.org/licenses/licenses.html#GPL

[3] OpenSPARC™ T2 Core Microarchitecture Specification[Z]. Revision A, Sun Microsystems, Inc., December 2007

[4] Firasta N, Buxton M, Jinbo P, et al. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency[Z]. Intel white paper, 2008

[5] The VIS™ Instruction Set V1.0[Z]. White paper, Sun Microsystems Inc., June 2002

[6] Makarov V N. The Integrated Register Allocator for GCC[C]// Proceedings of the GCC Developers' Summit. Ottawa, Ontario, Canada, July 2007: 77-90

[7] Stallman R M. The GCC Developer Community. GNU Compiler Collection Internals. For GCC version 4.6[Z]. Free Software Foundation, 2010